Design Pattern: It is at code level, like factory, Singleton

Architecture Pattern: Is overall layout, a high-level diagram, solution architecture like MVC, MVVM,

Architecture Principles/Style: Principles follow in the solution like Rest, SOA, IOC

Design patterns are best practices for object-oriented programming or solution to common software design problems.

Entities are objects or nouns. So identify entities and create classes and objects. Identify Pronouns and create properties of the object. And actions or verbs becomes method. There are two kinds of relationship between entities, ‘is a’ and ‘uses’. Shiv is a son of his father, it is a parent-child relationship. Shiv uses car or has a car, ‘uses’ relationship.

So for ‘is a ’ relationship you can use base class to point both type

|  |  |
| --- | --- |
| OOP Phase | Pattern |
| Template/Class creation | Structural |
| Class Instantiation | Creational |
| Runtime | Behavioural |

To test good software architecture look at how architecture handles changes.

Composition is flexibility and inheritance is structure, you have to balance both to get good software. If you go full composition, you will end up having set of randon things flying all over. And if you go full inheritance, you will end up having rigidity, you can’t change.

RIP pattern : Replace If with polymorphism, Instead of using if and creating instance based on condition. Use a dictionary to get instance of the base class type.

Lazy loading : A concept where we delay the loading of the object until the point where we need it. Load objects on demand rather than loading it unnecessarily. Like if a dictionary is needed then only add items in it. Don’t add it at declaration or in constructor.

Eager loading : Opp. Of lazy loading is eager loading.

**IOC (Inversion Of Control) :** It’s a principle where we move unwanted work or work which affects a class to some other entity, which takes that responsibility in more better way, is termed as inversion of control.

**Single Responsibility Principle :** A class should have single responsibility. It should not try to do lot of things.

**Separation Of Concern :** Unnecessary concern which are with the entity should be separated to some other thing

**Creational Pattern** It is all about different ways to create objects, helps to centralize object creation process and used by client to get objects. It prevents client to use concrete classes.

**Structural Pattern :** Relationship between objects. Suppose customer class is using phone class. So there is a relationship b/w customer and phone class. This pattern helps you solve the structural issues.

**Behavioural Pattern :** When you want to change the behaviour of an object and you don’t want this change affect the project very much.

All factory patterns help you control and centralize the object creational process. If you are creating same instance at many places, then for a single change, you have to modify it at all the places. So, by centralizing the object creational code, you just have to change it at one place.

**Simple Factory Pattern :** Centralize new keyword. It helps to centralize object creation and thus helps to achieve a decoupled system. Simply creating instance of a particular interface or base class without exposing the instantiation logic.

**Abstract Factory Pattern**, provide an interface for creating families of related or dependent objects. Here we add one more level to factory, and factory is created using a factory or we are abstracting a factory using a factory.

Abstract factory, concrete factory, abstract product, product

**Factory method pattern**, defines an interface for creating objects, but let subclasses to decide how to instantiate.

Product, concrete product, creator, concrete creator

**Builder Pattern,** separates the construction of a complex object from its representation so that the same construction process can be used for different representation.

Builder(abstract interface), concrete builder(actual construction code), Director(construct object using builder), Product(complex object)

Singleton : also known as anti pattern

**Behavioral Pattern**

**Strategy Pattern :**  It helps to choose algorithms dynamically. For ex. You can inject different validation algorithm to validate a class. It also helps you apply IOC, or SRP or SOC.